Course Application Design

Object-Oriented Design Principles

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Contents

- In this presentation, we’ll look at some fundamental OO design aspects:
  - Recap: the building blocks of Java
  - Polymorphism
  - Encapsulation
  - Abstraction

The building blocks of OO with Java

All code lives in classes

- In Java, all code lives in classes
- There are three base types:
  - class
  - interface
  - enum
- There are many variations in class implementations, and in the way these are linked into larger systems
- In this section we'll (re-)view the basic building blocks and techniques

Java POJO class

- The basic type is the Plain Old Java Class (POJO)
- It may contain a main() method
- It may subclass (extend) another class

A Plain Old Java Object POJO

Check out the package name, access modifiers and (im)mutability through constructor args and getters and setters

```java
package nl.bioinf.nomi.appdesigndemos.javabasics;

public class Address {
    private String street;
    private int number;
    private String zipCode;

    public Address(String street, int number) {
        this.street = street;
        this.number = number;
    }
    public String getStreet() { return street; }
    public int getNumber() { return number; }
    public String getZipCode() { return zipCode; }
    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }
}
```

Why getters & setters?

Using getters, you can restrict access and publish *virtual properties*

Using setters, you can restrict access and prevent data corruption

```
...
    private String zipCodeArea;
    private int zipCodeDetail;
...
    public String getZipCode() {
        return zipCodeArea + " " + this.zipCodeDetail;
    }

    public void setZipCode(String zipCode) {
        if (isCorrectZipCode(zipCode)) {
            this.zipCodeArea = zipCode.substring(0, 4);
            this.zipCodeDetail = Integer.parseInt(
                                           zipCode.substring(4).trim());
        } else {
            throw new IllegalArgumentException(
        "Given zipcode " + zipCode + " is not in correct");
        }
    }
    private boolean isCorrectZipCode(String zipCode) {
        return zipCode.matches("[a-zA-Z]{4} ?\\d{2}");
    }
}
```

Use Composition to build systems

Create small, maintainable and testable classes that conform to the Single Responsible Principle and use composition to build larger systems

```
public class Employee {
    private int employeeId;
    private Address address;
    private String name;
    private double salary;
...
}
```

A main() class

Usually, an application will have one class with a main() method. If you have a need for more, this is indicative your code should be split into separate jars.

```java
public class MyApp {
    public static void main(String[] args) {
        new MyApp().start();
    }

    private MyApp() {}

    private void start() {
        Logger.getLogger("MyApp.class")
               .info("Starting the analysis");
    }
}
```

Keep methods small
Have code readable without comments

```
public void setZipCode(String zipCode) {
    if (isCorrectZipCode(zipCode)) {
        parseZipCode(zipCode);
    } else {
        throw new IllegalArgumentException(”Wrong zipcode " + zipCode);
    }
}

private boolean isCorrectZipCode(String zipCode) {
    if (null == zipCode || zipCode.length() == 0) {
        return false;
    }
    return zipCode.trim().matches("\\d{4} ?[A-Za-z]{2}");
}

private void parseZipCode(String zipCode) {
    zipCode = zipCode.trim();
    int offset = 0;
    if (zipCode.contains(" ")) {
        offset = 1;
    }
    this.zipCodeArea = Integer.parseInt(zipCode.substring(0, 4));
    this.zipCodeLocal = zipCode.substring(4 + offset);
}
```

Can you spot the SRP violation?
The efficiency violations?

Keep methods small
Have code readable without comments

```java
public void setZipCode(String zipCode) {
    if (isCorrectZipCode(zipCode)) {
        parseZipCode(zipCode);
    } else {
        throw new IllegalArgumentException(”Wrong zipcode " + zipCode);
    }
}

private boolean isCorrectZipCode(String zipCode) {
    if (null == zipCode || zipCode.length() == 0) {
        return false;
    }
    return zipCode.trim().matches("\\d{4} ?[A-Za-z]{2}");
}

private void parseZipCode(String zipCode) {
    zipCode = zipCode.trim();
    int offset = 0;
    if (zipCode.contains(" ")) {
        offset = 1;
    }
    this.zipCodeArea = Integer.parseInt(zipCode.substring(0, 4));
    this.zipCodeLocal = zipCode.substring(4 + offset);
}
```

parseZipCode both parses and sets the zipCode elements.
Is this bad?
Mmmwa…
Can you improve the design?

OOP concepts

Interfaces and polymorphism are key to abstraction

Abstraction -the concept of **hiding the complexities of a system from the users of that system**- is key to almost all OO design aspects. It can be achieved through the use of interfaces or abstract classes.

An interface defines a contract that you can code against without knowing about the implementation details or technology

```
public interface EmployeeDataSource {
    Employee getEmployee(int employeeId);
    void storeEmployee(Employee employee);
}
```

Interface default methods

- Note that since Java 8, Interfaces can also provide –default- functionality on the condition that it is not dependent on a concrete instance.

```
public interface InterfaceWithDefault {
    default Foo createFoo() {
        return new Foo();
    }

    public static class Foo{}
}
```

Interface users know nothing about the implementer

Code against interfaces, not implementations

```java
public class EmployeePanel extends JPanel{
    private final EmployeeDataSource dataSource;

    public EmployeePanel(EmployeeDataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void showEmployee(int employeeId) {
        Employee emp = dataSource.getEmployee(employeeId);

        //GUI logic follows
    }
}
```

Abstract class: another type of interface

- Abstract classes use inheritance as a means to abstraction. They provide generic functionality but often leave some specifics unimplemented
- Here is a piece of the app that smells a bit (**code smell *Prefer Polymorphisms to If/Else or Switch/Case***)
- What if new types of employment are added? How many functions have employment-type specific code?
- Let’s refactor **`Employee`** using an abstract class

```java
public LocalDate getEndOfEmployment() {
    switch (this.commissionType) {
        case FREELANCE:
            return LocalDate.now();
        case PERMANENT:
            return getRetirementDate();
        case TEMPORARY:
            return LocalDate.now();
        default:
            throw new IllegalStateException(
                         "unknown constant:" + commissionType);
    }
}
```

Make Employee abstract

- Employee is now abstract. It has a a single abstract method: getEndOfEmployment(). The rest is concrete
- When a single method of a class is unimplemented, you **must** make it abstract
- A class with only concrete methods **may** be made abstract

```
public abstract class Employee {
...
//  private CommissionType commissionType = CommissionType.PERMANENT;
    public abstract LocalDate getEndOfEmployment();
```

Make concrete subclasses

PermanentEmployee, FreelanceEmployee, TemporaryEmployee are concrete subclasses and must implement the abstract method – and provide a matching constructor

```java
public class PermanentEmployee extends Employee {
    public PermanentEmployee(int employeeId,
                             Address address,
                             String name) {
        super(employeeId, address, name);
    }

    @Override
    public LocalDate getEndOfEmployment() {
        return null;
    }

}
```

Abstraction at package level

Suppose you have this `DataSource` contract, and several implementations. You could organize your package as follows:

package dao

- public interface EmployeeDataSource
- public class EmployeeDataSourceFactory
- <default> class EmployeeDataSourceFactoryMySQL
- <default> class EmployeeDataSourceFactoryDummy

This way, only the API is published, not the implementation (Of course, the Module system in Java extends this but is out of scope)

Abstraction at class level

In a class, make only those methods public that *really* should be visible to other classes. Always use the most stringent access level possible.

Abstraction vs Encapsulation

- Abstraction refers to the concept of hiding the complexities of a system from the users of that system
- Encapsulation is a language construct which bundles data and behavior together and restricts access to these (hides implementation details)
- Abstraction also hides, but *abstraction hides complexity*. Encapsulation, on the other hand, *hides the constructs it encapsulates*.

Polymorphism is the other pillar under abstraction

- No client of the Employee class needs to know which implementation it is talking to
- They are all Employee objects behaving slightly different, but always adhering to the Employee contract
- This is the essence of **Polymorphism** (and abstraction)

Constants

Enums -- constants on steroids

Do not use **public static final String CONSTANT**. They are not typo-resistant, and do not offer added values

```
public Date getEndOfEmployment() {
    switch(this.commissionType) {
        case FREELANCE: return new Date();
        case PERMANENT: return getRetirementDate();
        case TEMPORARY: return getTempContractEnd();
        default: throw new IllegalStateException(
                      "unknown type:" + commissionType);
    }
}
```

Testing

Test!

- Writing tests and executing these regularly
  - Makes your application reliable
  - Forces you to create *testable code*
  - Forces you to think about usage scenarios
  - Will save you the time from starting the application over and over and over
  - Will save you time debugging

Create test stubs

Create failing Junit test stubs for different scenarios:

- sunny day
- null/empty/exception scenarios
- border scenarios.

```
public class EmployeeTest {
...
    @Test
    public void getRetirementDateNormal() throws Exception {
        assertTrue( "Not implemented yet",false);
    }

    public void getRetirementDatePast() throws Exception {
        assertTrue( "Not implemented yet",false);
    }

    public void getRetirementDateNoBirthday() throws Exception {
        assertTrue( "Not implemented yet",false);
    }
}
```

Write test code – for a single method at a time

Create Junit tests for (all or the most important) methods of your app, before you actually implement them.

```java
@Test
public void setZipCodeDutch() throws Exception {
    this.address.setZipCode("1111AB");
    assertEquals("1111 AB", this.address.getZipCode());

    this.address.setZipCode("2222 CZ");
    assertEquals("2222 CZ", this.address.getZipCode());

    this.address.setZipCode(" 3333DQ ");
    assertEquals("3333 DQ", this.address.getZipCode());
}

@Test(expected = java.lang.IllegalArgumentException.class)
public void setZipCodeUS() throws Exception {
    this.address.setZipCode("90210");
}
```

…and then implement this single feature

- A JUnit test can cover only the non-private methods
- If written correctly, it will indirectly also cover the private members.
- If test coverage is really required, make methods package-visible (default or protected)

Adjust the tests

EmployeeTest should be adjusted because Employee cannot be instantiated anymore since it is abstract. Also, three distinct Employee subclass instances should be created

```
@Test
public void getEndOfEmploymentFreelance() throws Exception {
    assertEquals(LocalDate.now(), freelanceEmployee.getEndOfEmployment());
}

@Test
public void getEndOfEmploymentPermanent() throws Exception {
    LocalDate birthday = LocalDate.of(1969, 1, 20);
    permanentEmployee.setBirthday(birthday);
    LocalDate endOfEmployment = LocalDate.of(1969 +
Employee.RETIREMENT_AGE, 1, 20);
    assertEquals(endOfEmployment, permanentEmployee.getEndOfEmployment());
}

@Test
public void getEndOfEmploymentTemporary() throws Exception {
    assertEquals(LocalDate.now(), temporaryEmployee.getEndOfEmployment());
}
```

Make the tests pass again

As you can see, two subclasses of Employee have exactly the same implementation. When ManagerEmployee will be added, it will probably be the same as PermanentEmployee (maybe with some extra benefits though…)
This is another code smell that we will maybe later solve using the Strategy Design Pattern

```java
// public class PermanentEmployee extends Employee
@Override
public LocalDate getEndOfEmployment() {
    return getRetirementDate();
}
```

```java
// public class FreelanceEmployee extends Employee
@Override
public LocalDate getEndOfEmployment() {
    return LocalDate.now();
}
```

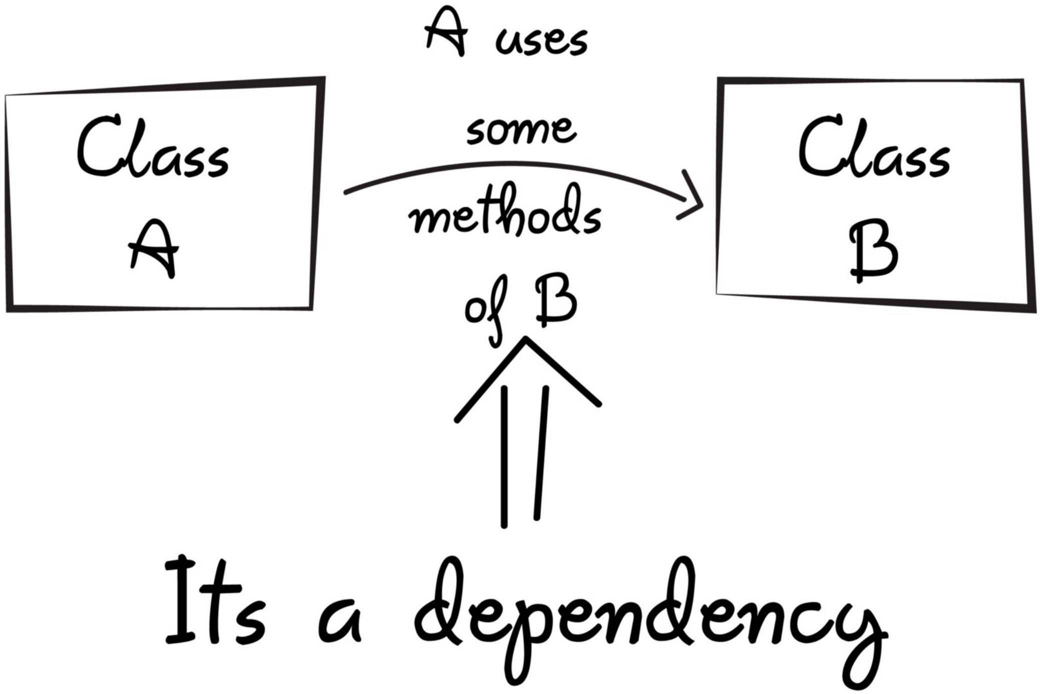
```java
// public class TemporaryEmployee extends Employee
@Override
public LocalDate getEndOfEmployment() {
    return LocalDate.now();
}
```

Dependency injection

example from

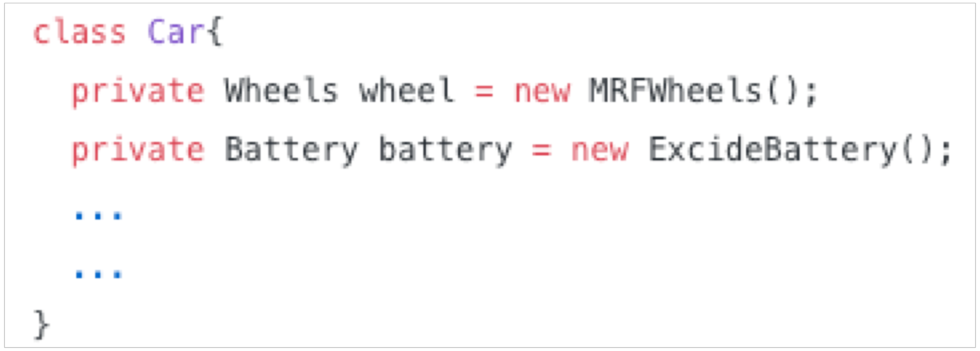
https://medium.freecodecamp.org/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f

Dependency



Why Dependency Injection?

- Let’s say we have a car class which contains various objects such as wheels, engine, etc.



```
class Car{
   private Wheels wheel = new MRFWheels();
   private Battery battery = new ExcideBattery();
   ...
   ...
}
```

- The car class is responsible for creating all the dependency objects.
- Now, what if we decide to ditch **MRFWheels** in the future and want to use **Yokohama** Wheels?

- We will need to recreate the car object with a new Yokohama dependency.
- When using dependency injection (DI), we can change the Wheels at runtime (because dependencies can be injected at runtime rather than at compile time).
- You can think of DI as the middleman in our code who does all the work of creating the preferred wheels object and providing it to the Car class.

Three ways to DI

- **constructor injection:** the dependencies are provided through a class constructor.
- **setter injection:** the client exposes a setter method that the injector uses to inject the dependency.
- **interface injection:** the dependency provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency.

Two ways to DI

```
class Car{
    private Wheel wheel;
    private Battery battery;

    // Constructor Based DI
    Car(Wheel wh, Battery bt) {
        this.wheel = wh;
        this.battery = bt;
    }

    // Setter Based DI
    void setWheel(Wheel wh){
        this.wheel = wh;
    }

    // Rest of code
}
```

DI in Spring

- Spring is a framework that has DI at its heart
- Here is the same example ported to Spring
- The Spring framework will take care of instantiating and “wiring” all the “components”

DI in Spring

```
@Component
class Car {

    @Autowired
    private Wheel wheel;
    private Battery battery;

    @Autowired
    Car(Wheel wh, Battery bt) {
        this.wheel = wh;
        this.battery = bt;
    }

    // Rest of code
}
```

DI: formal

- Dependency injection is a technique whereby one object supplies the dependencies of another object
- A dependency is an object that can be used (a service).
- An injection is the passing of a dependency to a dependent object (a client) that would use it.
- The service is made part of the client's state.
- Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Summary

Summary basic OO principles

- Encapsulate what varies
- Favor composition over inheritance
  - A class can only subclass one other (IS-A), but can hold many (HAS-A)
- Program against interfaces, not implementations
  - Here, an interface can be: Interface or (abstract) superclass

OO basics: abstraction

- Abstraction is the process of hiding details of implementation in programs and data, usually through the use of classes.
- Abstraction is therefore the process of separating ideas from specific instances of those ideas at work

OO basics: encapsulation

- Encapsulation is the packing of data and functions into a single component. The features of encapsulation are supported using classes.
- Classes allow selective hiding of properties and methods in a class by building a wall to protect the code from accidental corruption

OO basics: polymorphism

- Polymorphism is the provision of a single interface to instances of different types
- There are several different kinds of polymorphism:
  - **Ad hoc polymorphism** using function overloading: a function shows different implementations depending on a limited range of individually specified types and combinations
  - **Generics (parametric polymorphism)**: If the code is written without mention of any specific type and thus can be used transparently with any number of new types
  - **Inheritance polymorphism**: a name may denote instances of many different classes as long as they are related by a common superclass

- That finishes the Java basics review
- Check out the version with tag 0.1.1 of the AgileAppeDesign repo at https://bitbucket.org/minoba/applicationdesignmaterial/ if you are interested in the code so far